UNITED STATES PATENT APPLICATION

OF

DR. JÜRGEN KIENHÖFER AND RANJIT DESHPANDE

FOR

METHOD AND APPARATUS FOR EXECUTING MULTIPLE JAVA(TM) APPLICATIONS ON A SINGLE JAVA(TM) VIRTUAL MACHINE

PREPARED BY WILSON SONSINI GOODRICH & ROSATI

RELATED APPLICATIONS

This application relates to, incorporates by reference, and claims priority from, United States Provisional Patent Application Serial Number 60/112,803 entitled, "Method and Apparatus for Executing Multiple JAVA Applications on a Single JAVA Virtual Machine", filed December 18, 1998, having inventor Dr. Jürgen G. Kienhöfer.

BACKGROUND OF THE INVENTION

Field of the Invention

This invention relates to the field of improved execution environments for software applications running in the JAVA(TM) language. In particular, the invention relates to improvements designed to support the operation of multiple unmodified JAVA(TM) applications on an unmodified JAVA(TM) Virtual Machine.

Description of the Related Art

1. Description of Problem

JAVA(TM) applications are transportable byte codes that can be executed on a number of platforms. The execution environment for JAVA(TM) applications is a JAVA(TM) Virtual Machine (JVM). For each platform a JVM must be available to execute the JAVA(TM) applications. The specification of, and the implementation of, a JVM is described in documents such as "JAVA(TM) Virtual Machine", John Meyer and Troy Downing, O'Reilly and Associates, 1997. Additionally, the book "The JAVA(TM) Virtual Machine

Specification", Tim Lindholm and Frank Yellin, Addison Wesley, 1997, also describes a JVM.

For each JAVA(TM) application that a user wishes to run on a JAVA(TM) Virtual Machine, a separate JVM must be run together with a separate execution environment. This execution environment includes an object store in memory, also referred to as a JAVA(TM) heap, for storing JAVA(TM) objects and data. Also, the environment includes a "garbage collector" that deletes unused objects and compacts the JAVA(TM) heap. This arrangement consumes large amounts of system memory or each JVM, and thus each application. When used, as JAVA(TM) was originally designed on a client computer, this is not a problem as a user is typically running only one or two applications at a time. Further, the user's computer is dedicated to running those applications.

In contrast, server computers may require that multiple JAVA(TM) applications be running simultaneously. For example, a server might be deployed to handle processing for a large number of client computers. If the JAVA(TM) standard is followed, each of running JAVA(TM) application on the server would need to run in separate JVM's, each with an associated execution environment. Thus, each application would have its own JAVA(TM) heap and separate garbage collection process. The multiplicity of the garbage collection processes across all of the JVM's can consume significant amounts of processor time and lead to decreased performance.

The JAVA(TM) Virtual Machine could be rewritten in its entirety to support multiple applications at one time. However, such an approach would

require major re-architecting of the JAVA(TM) and/or JVM standards and specifications. Additionally, JAVA(TM) applications could be rewritten in source code to support multiple applications on a single JVM.

2. Prior Art JAVA(TM) Execution Environment

Turning to Figure 1, and the prior art JAVA(TM) execution environment, the execution environment includes a JAVA(TM) Virtual Machine 100, including JAVA(TM) base classes 102 and a primordial class loader 104. An optional application class loader 106 is depicted as well as a single JAVA(TM) application 108. This is the typical JAVA(TM) execution environment according to the prior art.

In the normal operation of a JVM, a JAVA(TM) application compiled to run on the JVM arrives in a sequence of byte codes arranged in class files. The class files, which can be remotely or locally accessed, are loaded by class loaders and executed in a threaded environment by the JVM. Execution of JAVA(TM) applications take place in threads, which are part of thread groups, and invokes calls to methods of the associated objects. Each thread that is created from a thread in a given thread group also belongs to that thread group. Executions of thread causes the creation of objects, which are stored in portions of the JAVA(TM) heap during run time. An application is able to run on a JAVA(TM) execution environment with a large set of JAVA(TM) base classes, which are sometimes considered part of the JVM. The base classes received calls from the application to enable many basic functions.

Object creation during execution within the JVM utilizes a class loader architecture. There are two types of class loaders in the JAVA(TM) execution

environment. The first type is a "primordial" class loader, e.g. the primordial class loader 104. The primordial class loader is considered part of the JVM itself and is designed to load certain class loaders. Usually the primordial class loader 104 is used to load classes of the application.

Another type of class loader is available for loading objects. This type of class loader is a JAVA(TM) class loader object written in JAVA(TM). This type of class loader can be installed by a JAVA(TM) application into a thread. When this type of class loader is installed into a thread other application objects within that thread are loaded using that class loader.

Notably the prior art JVM does not allow for a hierarchy of application class loaders. Thus, a JAVA(TM) application such as the JAVA(TM) application 108 cannot install additional application class loaders.

3. Conclusion

The prior techniques do not permit the execution of multiple unmodified JAVA(TM) applications on a single unmodified JAVA(TM) Virtual Machine. Further, the prior techniques do not support shared usage of the JAVA(TM) base classes by applications running on the JVM. Accordingly, what is needed is a method and apparatus for supporting multiple unmodified JAVA(TM) applications on a single JVM using a single copy of the base classes for all of the JAVA(TM) applications.

SUMMARY OF THE INVENTION

A modified JAVA(TM) execution environment is described. The modified environment supports multiple JAVA(TM) applications on a single

JAVA(TM) virtual machine (JVM). This modified environment provides significant memory and performance improvements when running multiple applications on a single computer system. Notably, no changes are needed to the source code of an application to take advantage of the modified environment. Further, embodiments of the invention may support shared access to base classes through the use of overlays. Additionally, system resource permissions can be enforced based upon the user permissions associated with a running application. Notably, embodiments of the invention allow multiple applications to share the abstract window toolkit (AWT) on a per display basis. Since only a single garbage collection routine is necessary, applications see improved performance relative to running in different JVMs. Further, the shared base classes eliminate significant memory overhead.

According to some embodiments of the invention, a class loader and a thread group is dynamically generated for each application to be run in the modified environment. This class loader defines a name space for the application. Additionally, the thread group defines the set of threads for that application. The class loader also loads the application classes into the JVM. The application itself can have an application class loader, an application security manager, and/or create additional thread groups.

As necessary, when the overlaid base classes are called, the calling application can be determined. Two approaches are used by some embodiments of the invention. In the first approach, the class loader of the calling method is determined. This in turn allows the identification of the application through reference to data associated with the class loader. Another approach is to scan

the thread group hierarchy of the JVM to identify a thread group with which application information has been associated by the class loader. Once the application is identified, the underlying base class functionality can be implemented using the appropriately selected files, resources, variable values, etc.

Additionally, embodiments of the invention can support system resource permissions, e.g. user access rights, on a per application basis. Each application can be associated with a user. The overlaid base

BRIEF DESCRIPTION OF THE FIGURES

Fig. 1 illustrates a JAVA(TM) Virtual Machine environment according to the prior art.

Fig. 2 illustrates a JAVA(TM) Virtual Machine environment according to one embodiment of the invention.

DETAILED DESCRIPTION

The modified JAVA(TM) execution environment supported by embodiments of the invention will be described with reference to Figures 1 and 2. Figure 1 shows the JAVA(TM) execution environment according to the prior art and was described above. Figure 2 shows the JAVA(TM) execution environment according to one embodiment of the invention and will now be described in greater detail.

Figure 2 shows the modified JAVA(TM) execution environment according to one embodiment of the invention. Elements of figure 2 that are found in figure 1 are designated with the same reference numerals. For example, Figure 2 includes the JVM 100. The JVM 100 used in Figure 2 may be identical to the JVM 100 used in Figure 1, but should at least be a substantially unmodified JVM.

The term "substantially unmodified" as used in this application refers to a JVM or JAVA(TM) application suitable for use in the prior art JAVA(TM) execution environment of Figure 1. For example, a JVM supporting just in time (JIT) that can execute substantially unmodified JAVA(TM) applications would be a substantially unmodified JVM. One further example may be instructive. A JAVA(TM) application 108 is substantially unmodified, if it can be used in the execution environment of Figure 1 without the need for source code – or byte code – modifications to run in the execution environment of Figure 2.

Examples of substantially unmodified JVMs usable according to embodiments of the invention include JVMs from Sun Microsystems, Mountain View, California; JVMs from Microsoft Corporation, Redmond, Washington;

JVMs from Apple Computer Corporation, Cupertino, California; and/or other available JVMs. For the purposes of this discussion it will be assumed that the JAVA(TM) Virtual Machine complies with a JAVA(TM) standard and that the JAVA(TM) applications similarly comply with a JAVA(TM) standard.

The elements of Figure 2 will now be described in greater detail. The substantially unmodified JVM 100 supports the modified execution environment of Figure 2. The substantially unmodified JVM 100 includes base classes 102. The base classes 102 are substantially unmodified base classes suitable for use in a standard JAVA(TM) execution environment such as the JAVA(TM) execution environment of Figure 1.

Additionally, Figure 2 includes base class overlays 200. The base class overlays 200 provides support for multiple JAVA(TM) applications using only a single copy of the base class 102. The base class overlays 200, allow multiple applications to reference the base classes 102 without conflicts due to different access privileges and/or base class definitions that inhibit sharing. The base class overlays 200 will be described in further detail below. The modified JAVA(TM) execution environment also includes a primordial class loader 104 that is substantially unmodified and suitable for use according to the prior art JAVA(TM) execution environment.

The modified JAVA(TM) execution environment includes a multiple application class loader 206. This class loader provides support for multiple applications. Additionally, a security manager 204 provides for different degrees of access to different applications based on privileges. The multiple

application class loader 206 handles the class loading of JAVA(TM) applications within the modified execution environment of Figure 2.

Compare this to the standard execution environment of Figure 1, where the primordial class loader 104 would load the JAVA(TM) application 108. According to the modified execution environment of Figure 2, the multiple application class loader 206 would invoke the JAVA(TM) application 108.

In order to support the launch of multiple applications, a launch interface 202 is provided. The launch interface 202 may itself be a JAVA(TM) application. The launch interface 202 may provide a command line interface or other interface for invoking the execution of JAVA(TM) applications within the modified execution environment of Figure 2. The launch interface 202 may itself be loaded by the multiple applications class loader 206 as a JAVA(TM) application running within the modified JAVA execution environment. In some embodiments, the launch interface may respond to remote procedure invocations, or some other type of message, and execute applications according to parameters specified in the message. In some embodiments, the launch interface 202 provides a UNIX-style command line interface with log in and security procedures.

The depiction of the multiple application class loader 206 as a single class loader for all applications is a simplification. In fact, a class loader is dynamically generated for each application. Thus, the launch interface 102, the JAVA(TM) application 108, and the JAVA(TM) application 108B each has a dynamically generated multiple application class loader 206 responsible for loading the appropriate application classes. Each of the dynamically generated

multiple application class loaders can define its own namespace within which the loaded applications will execute.

As shown in Figure 2, once invoked from the launch interface 202, JAVA(TM) applications (e.g. the JAVA(TM) application 108) can have their respective classes loaded within the modified execution environment of Figure 2. Similarly a second application, the JAVA(TM) application 108B could be loaded within the modified execution environment of Figure 2. These two applications would be sharing the same JAVA(TM) Virtual Machine 100 and the same base classes 102. However, the multiple application class loader 104 would place them in separate namespaces and would place them in different thread groups. The base class overlays 200 ensure appropriate behavior of the base classes 102 for each of the applications.

Notably, neither the JAVA(TM) application 108 nor the JAVA(TM) application 108B need to be modified at the source code level to operate within the modified execution environment. The environment of Figure 2 is transparent to JAVA(TM) applications running in the environment.

The modified execution environment of Figure 2 only needs a single garbage collection process and a single copy of the base classes 102. This provides significant memory and speed savings. In some experiments, this reduced the memory overhead to enable execution of over one hundred JAVA(TM) applications on a single JVM – on a single server computer, which would otherwise support only fourteen of these applications at the same time. See also, Jürgen G. Kienhöfer, "Java Junction: Perkup, SCO Server Side Java

Technology", in SCO Coredump, Summer 1999, Number 13, page 8, also available at <http://www.sco.com/developer/core13/perkup.htm>.

The security manager 204 is an addition to the inherent security models of JAVA(TM). Prior art JVMs were typically invoked on client machines by a specific user and the single application ran with that user's privileges. In contrast, the execution environment of Figure 2 would typically be invoked with system privileges such as "root" on UNIX-like systems. As a result, each running application would, without additional security, be capable of accessing the entire system. Therefore, a security manager 204 can be provided to enforce operating system security – or other security – requirements.

In one embodiment of the invention, the security manager 204 uses parameters provided via the launch interface 202 to control the permissions granted to running applications. For example, if using the launch interface 202 an application is invoked using the privileges of "user 1", the security manager 204 would enforce operating system file permissions and resource permissions for that application according to the privileges granted to "user 1". Examples of enforced requirements include those for:

- reading, writing, creating, deleting, modifying, or examining system resources such as files and sockets;
- listening or accepting network connections to a reserved port;
- executing a program on the system or starting a sub process
- terminating the JAVA(TM) run time environment of Figure 2
- loading dynamic libraries and native methods.

Thus if the permissions on a particular file "x" indicate that it is owned by "user 2" and not readable by other users, an attempt by a JAVA(TM) application running as "user 1" to read the file may be denied.

Each application running in the environment of Figure 2 may also have its own security management policies — for example, a set of JAVA(TM) sandbox policies.

Part of the base class overlays 200 involves the separation of certain resources that are not effectively shared between different programs. For example, the standard java.lang.system class uses static variables to define the input, output and errors streams. As a result, it is not possible to share that class without modification of the base classes. In this instance, the base class can be overlaid with modifications that can use two possible processes – possibly in conjunction with one another - to determine the current application and provide appropriate access to the shared base class.

One process used by overlaid, or shared, classes to identify the correct application is to identify the class loader for the calling thread. If the class loader is an instance of the dynamically generated multiple application class loader, then it together with the namespace can be used to identify the application. Consequently, the correct resource permissions, list of accessible files, input and output devices, etc., are identified for use by the shared class.

The above approach may fail if the class loader for the object accessing the overlaid class is the primordial class loader 104. In that instance, the associated namespace may not provide adequate information to suitably identify the application and needed information.

Therefore, a second approach to determining the calling process can be used. In this case, the thread data structures within the JVM 100 can be examined to determine the calling object's thread. Then, the group for the thread can be identified. Information associated with the thread group about the application and its properties can then be identified. If necessary, the thread group hierarchy can be recursively examined until a thread group is found that is associated with information about the process.

As seen in the execution environment of Figure 2, the multiple application class loader 206 does extend the JVM 100 to support application class loaders in addition to the multiple application class loader 206. In some embodiments, it is necessary to configure the JVM not check for multiple class loaders to enable this capability. In other embodiments this change is not necessary if the JVM itself already supports hierarchies of class loaders.

The base class overlays 200 may involve adding checks for resource permissions. For example, the procedures for reading file must be overlaid to include identification of the user for the application, as described above, as well as verification of the user's rights with respect to that file. These changes to the base classes may be implemented in the base class overlays 200, in the security manager 204, or in a combination of the two.

The terms "program", or "computer program", as used in this application, refers to any sequence of instructions designed for execution on a computer system. A program may include a subroutine, a function, a procedure, an object method, an object implementation, an executable application, an

applet, a servlet, a source code, an object code, and/or some other sequence of instructions designed for execution on a computer system.

The base class overlays 200, the multiple application class loader 206, and the security manager 204 may be embodied as one or programs included in one or more computer usable media such as CD-ROMs, floppy disks, or other media.

Some embodiments of the invention are included in an electromagnetic wave form. The electromagnetic waveform comprises information such as base class overlays, a multiple application class loader, and a security manger for use in a modified JAVA(TM) execution environment. The electromagnetic waveform may include the multiple application class loader accessed over a network.

The foregoing description of various embodiments of the invention has been presented for purposes of illustration and description. It is not intended to limit the invention to the precise forms disclosed. Many modifications and equivalent arrangements will be apparent.